Справочник для начинающего по Forth

by
J.V. Noble

Содержание

Вступление
Начало работы
Структура Форта
Расширение словаря
 Стеки и обратное примечание (RPN)
    Управление параметром в стеке
    Результат стека и его применение
    Локальные переменные
Использование памяти
 Сравнение и переход
Документация и комментарий кода
    Использование замечаний
    Комментарии стека
    Наклонная линия (\)
    Блоки комментариев
    Самодокументация кода
Встроенные арифметические операции
Выполнение цикла и структурное программирование.
     Неопределенные циклы
    Определенные циклы
    Структурное программирование
    "Нисходящий" метод
CREATE ... DOES> (жемчужина Форта)
     Определение слов "определения"
    Время выполнения против действий во время компиляции
     Важность данных (встроенные модули)
    Расширенное использование компилятора
Арифметика с плавающей точкой
Нетривиальный пример программирования
Некоторые замечания о строках
    Алфавитно-цифровые строки
    Числовой Вывод
Некоторая полезная справочная информация

1. Вступление

Форт - необычный машинный язык, который применяется к различным проектам. Это - очевидный выбор, когда важны такие фокторы как, скорость выполнения, компактность кода, или любой комбинации вышеупомянутого.

Его также назвают "... один из наиболее секретных в компьютерном мире." Это не приувеличение: большие корпорации приобрели Форт системы, разработки программ, от продавцов, типа Laboratory Microsystems, Inc, Forth, Inc или MicroProcessor Engineering, Ltd.

Некоторые сплетничают (недоброжелательно) что корпоративные гиганты предпочитают скрывать их позор при использовании Форта; но я полагаю, что они фактически скрывают «секретное оружие» от их конкурентов.

Всякий раз, когда Форт конкурировал непосредственно с более известным языком C, он без труда побеждал, по производительности кода, по быстроте, более надежный код. Я искал примеры с противоположным результатом, но не смог найти даже одного случая.

2. Начало работы

Мы будем использовать Win32Forth для этих иллюстраций. Загрузите файл

    w32for42.exe

двойной клик на нём, приведёт к установке на любую Windows 95,-98,-NT, - ME или-XP – оборудованную машину.Сжатые файлы разархивируются.Они должны также установить группу программ на вашем настольном компьютере.Теперь запустите Win32Forth,открывая группу программ и нажимая на соответствующий значок.

В результате должно открыться окно, содержащие что-то типа:

```
32bit Forth for Windows 95, and NT
Compiled: July 23rd, 1997, 5:11pm
Version: 3.5   Build: 0008   Release Build
Platform: Windows 95 Version: 4.0    Build: 16384
491k bytes free
2,719 Words in Application dictionary
1,466 Words in System dictionary
4,185 Words total in dictionaries
8,293 Windows Constants available

Loading Win32For.CFG
*** DON'T PANIC, Press: F1 NOW! ***
```

Вы можете использовать ВЕРХНИЙ или нижний регистр, чтобы печатать команды и данные. Win32Forth не чувствителен к регистру. Вы можете также запустить редактора WinView, нажимая на его значок в группе программ. Особенности редактора довольно стандартны и очевидны. Удостоверьтесь в установке персональных настроек (Ctl-Shift-P или используйте раскрывающееся меню Редактирования). Между прочим, WinView написан в Форте (полный исходник включен), и объединен с Win32Forth так, чтобы возникающие ошибки отображались в редакторе.

3. Структура Форта

В окне Win32Forth, теперь напечатайте

```
BYE    <cr>
```

Окно Win32Forth немедленно закроется.

Что случилось? Форт - интерактивный язык программирования, состоящий полностью из подпрограмм, вызывающий слова,говорят на жаргоне Форта. Интерактивный означает, что
Вы печатаете запрос на клавиатуре, и машина отвечает.
Детально Мы будем рассматривать, как это происходит ниже. Слово выполняется (в интерактивном режиме),при его вызове.

Мы только что наблюдали, как это происходит: BYE – подпрограмма Форта, означающая "выход в операционную систему". Таким образом то, когда мы напечатали BYE <CR> BYE, выполнялось, и система возвратила управление Windows.
Нажмите на значок Win32Forth снова, чтобы перезапустить Форт. Теперь мы будем пробовать кое-что более сложное. Введите

```
2 17   +   .   <cr> 19   ok
```

Что случилось? Форт является интерпретирующим. Маленькая программа называемая "внешний интерпретатор" непрерывно циклами, ждущими ввода от клавиатуры или от запоминающего устройства. Ввод – последовательность текстовых строк (слова или числа) отделенных друг от друга стандартным разделителем Форта: один или более пробелов ASCII (32decimal = 20hex) символы.

Текстовые строки могут интерпретироваться только тремя способами: слова (названия подпрограммы),числа, или "неопределенный".

Внешний интерпретатор пробует сначала искать поступившее слово в словаре (список уже-определенных названий подпрограммы). Если это слово найдено, то интерпретатор выполняет соответствующий код.
Если в словаре не существует такого слова, то интерпретатор пробует читать его как число.
Если строка удовлетворяет правилам, определяющия числа, то происходит преобразование к числу во внутреннем представлении машины, и сохранено в специальном местоположении памяти, названном "вершина стека" (TOS).

В вышеупомянутом примере, Форт интерпретировал 2 и 17 как числа, и поместил их обоих в стек.

"+" является предопределенным словом, как и ".", таким образом они находились и выполнялись
"+" прибавил 2 к 17 оставил 19 в стеке.
Слово "." (называемое "точкой") удалило 19 из стека и вывело
на стандартное устройство вывода (в нашем случае, монитор).

Диаграмма ниже - блок-схема, представляющая действия, выполненные Фортом внешним интерпретатором в течение интерпретации

Мы могли так же написать

```
HEX     0A   14   * . <cr>   C8 ok
```

(Вы поняли это? Подсказка: DECIMAL означает “перевести в десятичную систему счисления”, принимая во внимание HEX стандарты для “перехода к десятично-шестнадцатиричной системе счисления”.)

Если поступающий текст не может быть не расположен в словаре, не интерпретироваться как число, то Форт выдаёт сообщение об ошибке. Пробуйте это: напечатайте X <cr> и получите

```
X
Error: X не определён
```

Или напечатайте THING <cr> и получите

```
THING
Error: THING не определён
```

В заключение, программа "Hello, World!". Форт позволит вам Вывести текст используя слово ." остальное (мы рассмотрим в §4 ниже

Рассмотрим что : и ; означает):

```
: hi ." Hello, World!"  ;    ok
```

Теперь введите hi и посмотрите что случиться:

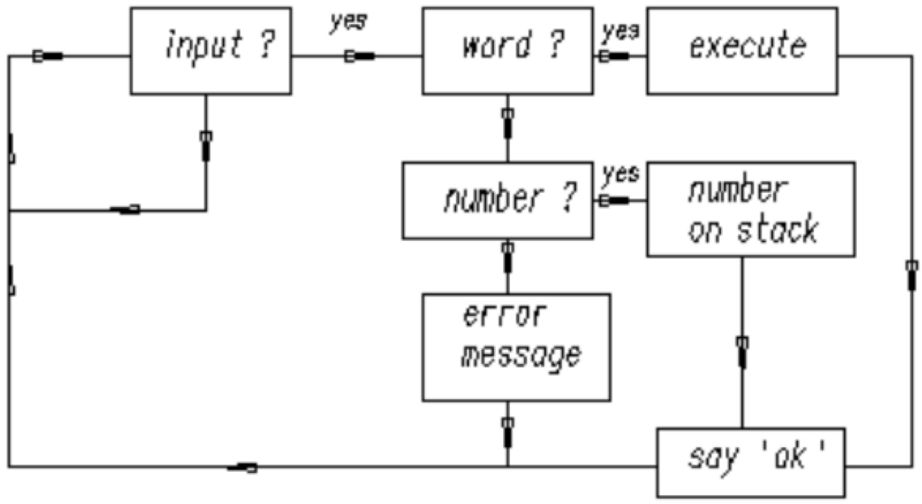
```
hi Hello, World! ok
```

Это может быть разработано со словами табуляции, переводы каретки, вывода на экран, и т.д. (Слово . " означает “Отображать строку, следующую за открывающейся кавычкой и заканчивающуюся " на стандартное устройство вывода.")

Форт принадлежит классу Последовательно интерпретируемых Языков. Это означает, что он
может интерпретировать команды (подпрограммы или программы) напечатанный в пульте, так же как создать (собирают) новые подпрограммы и программы. Компилятору на традиционном языке показали структуре ниже:

Чтобы собирать и проверять программу на традиционном языке, типа ФОРТРАНа, С или Паскаля, каждый готовит входной (исходный) файл, представляет это "черному ящику " что кто-то еще созданный (компилятор) и затем выполняет получающийся исполняемый файл (который находится вообще на машинном языке). Этот процесс может быть настолько утомительным, что большинство развития программы на традиционных языках должно быть поддержано сложным набором программ, названных "средой", состоя из интегрированных редакторов, отладчиков, каталогов управления версии и т.п.
Внешний интерпретатор/компилятор

Forth система похож на это:



Forth имеет немного общего с традиционным методом компиляции. Хотя Forth интерпретатор/компилятор изобразил схематически выше усложненных просмотров, это - простота непосредственно по сравнению с содержанием

представление традиционного компилятора черного ящика.

Непрерывная петля ждет входа - от клавиатуры, дискового файла или безотносительно - и действует на это согласно его природе. Вход состоит из последовательности слов и чисел.

Если название признано, это выполнено; если бы это не находится в словаре (где еще Вы держали бы список слов?) Forth пробует преобразовать это в число и выдвинуть это на стеке.

Если это невозможно, Forth прерывает выполнение, выпускает ошибочное сообщение и ждет входа.

Поскольку мы будем видеть ниже, что делает Дальше компилятор, так же как переводчик - набор слов (Forth подпрограммы), что, когда они напечатаны и казнены, создают новый Forth подпрограммы.

4. Расширение словаря

Компилятор - одна из наиболее покоряющих особенностей Forth. В отличие от большинства других языков высокого уровня, Forth компилятор - часть языка. (Язык обработки списков Лисп и его диалекты также делают компоненты механизма компиляции доступными программисту.) таким образом, его компоненты - Forth слова, доступные для программиста, который может использоваться, чтобы решить его проблемы.

В этой секции мы обсуждаем, как компилятор расширяет словарь. Как отмечено выше, обычно Forth система ждет вход, и интерпретирует (и выполняет это). Мы говорим,
что система находится обычно в, интерпретируют способ.

Forth использует специальные слова для создания новых словарных статей, то

есть новые слова. Наиболее важное ":" (“начинается новое определение”) и ";" (“заканчивается определение”).

Давайте попробуем: введем

```
: *+      *   +   ;   <cr>   ok
```

Что произошло? Слово ":" было выполнено, потому что уже было В словаре. Действие ":" -

> Создайте новую словарную статью *+ и включите из interpret в режим compile.

> Собирают способ, переводчик ищет слова, и - вместо того, чтобы выполнять их - устанавливает указатели на их кодекс. (Если текст – число, вместо того, чтобы выдвинуть это на стеке, Forth встраивает число в место словаря, выделенное для нового слова, после специального кода, который помещает сохраненное число в стек всякий раз, когда то слово выполнено.)

> Действие "*+" выполняется соответсвенно предыдущее определенным словам "*" и "+".

Теперь попытаемся *+ :

```
DECIMAL       5 6 7 *+ .     <cr>   47   ok
```

Этот пример иллюстрировал два принципа Forth: добавление нового слова к словарю, и испытанию это, как только это было определено.

Диаграмма ниже - блок-схема, представляющая действия, выполненные Forth внутренним переводчиком в течение компиляции.

Any word you have added to the dictionary can be decompiled using

the Forth word SEE. Say

```
SEE *+   <cr>
```

and get

```
SEE *+
: *+     * + ;   ok
```

This can be useful when trying to understand how something works.

5. Стеки и обратное примечание (RPN)

Мы теперь обсуждаем стек и "обратный польский язык" или "постустанавливаем" арифметику, основанную на этом. (Любой, кто использовал калькулятор Hewlett Packard,
должен быть знакомым с понятием.) Фактически весь современный центральный процессор
разработаны вокруг стеков. Forth эффективно использует его центральный процессор, отражая эту основную архитектуру стека в его синтаксисе. Но что такое является стеком?

Поскольку название подразумевает, стек - сборник изречений машины - регистрация груды карт с числами, написанными на них. Числа всегда добавляются к вершине груды, и
удаляются от вершины груды. Forth линия входа

```
2 5 73 -16 <cr> ok
```

Покидает стек в состоянии

```
cell #     contents
  0         -16          (TOS)
  1          73          (NOS)
  2           5
  3           2
```

Мы обычно применяем базирующуюся нулевую относительную нумерацию данных Forth (такие как стек, массивы, таблицы т.д.)

Полагайте, что мы сопроводили входную вышеуказанную строку с строкой

```
+ - * . <cr> xxx ok
```

```
cell#   initial        +    -       *          .

  0         -16       57   -52      -104
  1          73        5     2
  2           5        2
  3           2                                    empty
                                                   stack
```

Операция "." отображает -104 на экране, оставляя стек пустым. То есть, xxx = -104.

a. Управление параметром в стеке

Forth системы включают, по крайней мере, два стека: стек параметра и стек возвращения. Система на основе стека должна обеспечить пути поместить числа в стек, удалять их, и перестраивать их заказ. Forth включает стандартные слова с этой целью. Помещение чисел на стеке легко: просто напечатайте число (или включите это на определении Дальше слово). СНИЖЕНИЕ слова удаляет число из TOS и продвигает все другие числа. (Так как стек обычно становится нисходящим в памяти,
простое СНИЖЕНИЕ - ly увеличивает указатель на TOS 1 ячейкой.

SWAP заменяет верхние два числа.

DUP дублирует TOS в NOS.

ROT вращает верхние три числа.

Эти действия показаны ниже

```
cell | initial | DROP    SWAP    ROT        DUP
  0  |   -16   |   73     73       5        -16
  1  |    73   |    5    -16     -16        -16
  2  |     5   |    2      5      73         73
  3  |     2   |           2       2          5
  4  |         |                             2
```

Forth включает слова OVER, TUCK, PICK и ROLL, которые показаны ниже:

```
cell | initial | OVER    TUCK     3 PICK   3 ROLL
  0  |   -16   |   73    -16         2        2
  1  |    73   |  -16     73       -16      -16
  2  |    5    |   73    -16        73       73
  3  |    2    |   5       5         5        5
  4  |         |   2       2         2
```

Clearly, 0 PICK то же самое, что DUP, 1 PICK синоним для OVER, 1 ROLL означает SWAP, и 2 ROLL означает ROT.

b. Результат стека и его приминение

Мы заметили выше той компиляции, устанавливает связи от слова запроса до

предварительно-определенного призываемого слова. Механизм редактирования – в течение выполнения - использует стек возвращения (rstack): адрес следующего слова, которое будет призвано помещен в rstack, так, чтобы, когда текущее слово сделано, выполняя, система знала, чтобы подскочить к следующему слову. (Это – так в больше всего, но не все Forth выполнение. Но все имеют стек возвращения, действительно ли они используют их для того, чтобы связать подпрограммы.)
В дополнение к служению в качестве бассейна адресов возвращения (так как слова могут быть вложены, адреса возвращения нуждаются в стеке, который будет надет) Rstack - то, где пределы... Конструкция ПЕТЛИ помещена. Пользователь может также хранить/восстанавливать к rstack. Это экс - вполне достаточно из использования компонента в цели кроме того, для которого это было разработано. Такому использованию
препятствуют для новичков, так как это добавляет специю опасности для программирования.
См. "Примечание предостережения" ниже. Чтобы хранить к rstack, мы говорим> R, и восстановить мы говорим R>. Слово R копирует вершину rstack к TOS. Почему используют
rstack, когда мы имеем совершенно хороший стек параметра, чтобы играть с? Иногда становится твердо прочитать кодекс, который выполняет com-plex гимнастика на стеке.
Rstack может уменьшить сложность. Альтернативно, ПЕРЕМЕННЫЕ - названный местоположениями –
обеспечивают, место, чтобы хранить числа-, типа промежуточного звена приводит к вычислению –
от стека, снова уменьшая гимнастику. Пробуйте это:

```
\ YOU DO THIS                   \ Исключение

VARIABLE X <cr>      ok         \ создает поименованную позицию хранения X;
                                \ X выполняется, оставляя свой адрес

3 X ! <cr>     ok               \ ! ("store") ожидает число,
                                \ адрес, хранит число
                                \ к которому обращаются

X @    . <cr>   3 ok            \ @ ("fetch") ожидает адрес, и
                                \ место его содержания в вершине стека.

   : RDROP     ( or R>DROP)          ( r: n -- )   R>   DROP   ;
```

c. Локальные переменные и значения

Я упоминал VARIABLEs выше VARIABLE как подпрограмму, чье действие возвратить адрес памяти ячейки, как в

```
VARIABLE x
x . 247496       ok    ( Это не должно быть этим адресом!)
-49 x !
x @ .     -49    ok
```

Значение широко использовано как гибрид переменной и константы (смотрите ниже). Мы определяем и инициализируем значение, как константу:

```
13 VALUE    thirteen        ok
```

Мы вводим новое VALUE как CONSTANT:

```
thirteen    .     13   ok
```

Тем не менее, мы можем изменить VALUE, как если было бы VARIABLE:

```
47   TO   thirteen     ok
thirteen    .     47   ok
```

Разумеется, слово TO также работает со словами определениями, заменяющей VALUE которая следует в вершине стека. VALUEs -
часть набора основного расширения ANS Forth.

ANS Forth также включает набор локальных расширений. Локальные обычно Динамические по своей природе (их память восстанавливается на выходе слова), хотя стандарт не настаивает на этом. Обычно использованный синтаксис LOCALS| a b c ... |, как в этом определении (из алгоритма):

```
: v+    ( a b c d -- a+c b+d)
    LOCALS| d c b a |
    a c +    b d +   ;
```

```
2 3 4 5 v+ .S [2] 6 8     ok..     ( .S displays the stack without
destroying it)
```

Важные вещи запомнить

> имена a, b, c ... могут быть любыми приемлемыми строками Forth;

> локальные имена означают свое текстовое определение;

> локальные переменные инициализированы из стека, как показано v+ выше, и как в следующем примере:

```
: test-locals     ( a b c -- )
      LOCALS| c b a |
      CR   ." Normal order: " a .        b .   c .
      CR   ." Stack order:       " c .    b .   a .
      13 TO a     14 TO b      15 TO c   \ how TO works
      CR ." Changed: " a . b . c
;    ok

3 4 5 test-locals

Normal order: 3 4 5
Stack order:      5 4 3
Changed:      13 14 15    ok
```

> локальное действие подоюно VALUEs, не как VARIABLEs, как выше Приведенных примерах;

> набор локальных расширений требует LOCALS| ... | для размещения (по крайней мере) 8 локальных имен.

6. Использование памяти

Поскольку мы только видели, обычные числа принесены по памяти к стеку ("усилие"), и сохранены! (склад). ожидает адрес на стеке и заменяет тот адрес его использованием содержания, например, фраза X! ожидает, что число (НОМЕРА) и адрес (TOS) хранит это в, и помещает число в местоположение памяти, упомянутое адресом, потребляя оба аргумента в процессе, как во фразе 3 X! Двойные числа длины могут подобно быть принесены и сохранены, D и D!, если система имеет эти слова.

Положительные числа, которые представляют характеры, могут быть помещены в характер ячейки размера памяти, используя C и C!. Это удобно для операций с вереницами текста, например, показывать на экране и ввод / вывод клавиатуры. Конечно, нельзя поместить числа в память или восстановить их, впрочем, без средства распределения памяти и назначения ярлыков к памяти, столь ассигнованной. Дальше подпрограммы СОЗДАЮТ и ВЫДЕЛЯЮТ - основные инструменты для того, чтобы отложить память и прилагать удобный ярлык к этому.

Поскольку мы будем видеть ниже, СОЗДАВАТЬ, делает новый вход словаря, как в

```
CREATE X
```

Здесь новый вход имеет название X, но оно не могло бы иметь "Joe" или что-нибудь другое. Новое имя является подпрограммой Forth, которая будет возвращать адрес следующего доступного пространства в памяти. Например,

```
CREATE X   ok
X . 247316   ok
HERE . 247316     ok
```

HERE – подпрограмма, которая возвращает адрес следующего доступного

Пространства — мы отмечаем, что это тот же самый адрес X, потому что нет Выделенного пространства. Мы можем исправить это

```
10 CELLS ALLOT    ok
```

и проверяя с

```
HERE . 247356     ok
```

Мы видим, как следующее доступное пространство было выделено как 40 байтов в памяти. (Каждая Ячейка содержит следовательно 4 байта или 32 бита в этой системе.) Другими словами, подпрограмма выделяет увеличения указателя HERE числом блоков адреса, которые вы сообщили для выделения.
Вы могли бы сказать

```
40 ALLOT
```

вместо

```
10 CELLS ALLOT
```

но последнее более портативное, потому что оно освобождает, чтобы проверить ваш код, если вы запустили это в системе с 64-битовыми или 16-битовыми ячейками.

Выполняя последоввательность

```
CREATE X    10 CELLS ALLOT
```

Мы отклонили достаточно, чтобы держать 10 32-битовых числа—например, Таблица или массив—которые могут ссылаться присваиванием. Если мы хотим Получить шестой элемент массива (первый элемент имеет индекс 0, Соответственно шестой имеет индекс 5) мы можем сказать

```
X   5 CELLS   +
```

вычисляется его адрес. Для того, чтобы видеть, как это работает, скажем

```
137   X 5 CELLS +      !   ok
```

чтобы сохранить целое в шестой позиции массива; затем

```
X 5 CELLS +   @    . 137   ok
```

извлекает и отображает это.

Используя инструменты, обеспеченные СОЗДАЮТ и ВЫДЕЛЯЮТ, мы можем изобрести любой вид структуры данных, которую мы любим. Это - то, почему Forth не обеспечивает
защиту структур данных, типа должны быть найдены на языках как C, Паскаль или ФОРТРАН.

Это слишком легко в Forth таможенному портному, которого любой вид данных структурирует, каждый желает. В секции на СОЗДАЮТ... ДЕЛАЕТ> ниже Вас, будет видеть, что Forth облегчит писать подпрограммы ("конструкторы"), которые создают таможенные структуры данных – который может даже включить кодовые фрагменты, которые делают полезные вещи.
Например,

CONSTANT - число, которое Вы не хотели бы изменить в течение выполнения программы.

Таким образом, Вы не хотите доступ к его местоположению памяти. Как тогда Вы получаете число, когда Вы нуждаетесь в этом? Вы упаковываете код для с местоположением хранения, так, чтобы, называя ПОСТОЯННОЕ Вы восстановили его содержание. Его использование

```
17 CONSTANT seventeen     ok
seventeen . 17    ok
```

7. Сравнение и переход

Forth позволяет вам сравнивать два числа в стеке, используя операторы сравнения ">", "<", "=" . Например,

```
2 3 > <cr> ok
```

оставляет 0 ("ложь") в стеке, потому что 2 не больше, чем 3.

И наоборот фраза

```
2 3 < <cr> ok
```

оставляет -1 ("истина") потому что 2 меньше 3.

Замечания: В некоторых Forths “истина” - +1, а не -1.

Реляционные операторы поглощают как аргументы и оставляют “flag”, чтобы показать, что случилось.

(Много одноместных реляционных операторов предложения Forths "0=", "0>" and "0<".
Эти, как вы уже могли догадаться, определяются независимо от содержания вершины стека положительное или отрицательное.)

Реляционные слова использованы для перехода и управления. Например,

```
: TEST        0 =   INVERT     IF   CR   ." Not zero!"     THEN    ;

0 TEST <cr>    ok          ( нет действия)
-14 TEST <cr>
Not zero!     ok
```

Вершина стека сравнивается с нулем, и оператор INVERT приложен К результирующему значению. Слово CR выпускает перевод строки (конец строки). Наконец, если вершина стека не равна нулю, IF поглощает значение и выполняет все слова до завершения THEN. Если вершина стека равна нулю, выполнение переходит к следующему после THEN слову.

Слово ELSE использовано в утверждении IF...ELSE...THEN: ненулевая Величина в вершине стека вызывает любые слова между IF и ELSE для выполнения, и слова между ELSE и THEN пропускаются. Нулевое значение производит противоположное поведение. Таким образом,

```
: TRUTH       CR    0 =   IF   ." false"    ELSE   ." true"
THEN    ;

1 TRUTH <cr>
true    ok

0 TRUTH <cr>
false    ok
```

С тех пор THEN использовано для завершения утверждения IF, а не в своем обычном значении, некоторые авторы Forth предпочитают название ENDIF.

8. Документация и комментарии кода Forth

Forth иногда обвиняется в том, что он "только написанный" язык, то есть некоторые жалуются, что Forth является загадочным. Это - действительно жалоба против бедной документации и нетелеграфных названий слова. Неудобочитаемость - одинаково недостаток плохо письменного ФОРТРАНА, ПАСКАЛЯ, С, и т.д. Дальше программисты предложений, которые позаботились инструменты для того, чтобы произвести ясный код.

а. Заключенные в скобки замечания

Слово (-левая круглая скобка, сопровождаемая местом - говорит "игнорирование все после текста до следующей правильной круглой скобки в потоке входа". Таким

образом, мы можем вкрапить объяснительные замечания в пределах определений двоеточия.

b. Комментарии стека

Специфическая форма введенного замечания описывает эффект слова на стеке. В примере рекурсивной петли (НОД ниже), комментарии стека - действительно вся необходимая документация. Глоссарии вообще объясняют действие слова с комментарием эффекта стека. Например,

```
( adr -- n)
```

Описывает слово @ ("fetch"): это сообщает @ ожидание найти адрес (adr) в стеке, и покинуть его (n) по завершении.
Соответствующий комментарий для ! может быть

```
( n adr -- ) .
```

c. Наклонная черта (\)

Слово "\" (наклонная черта влево, сопровождаемая местом) недавно получило пользу как метод для включения более длинных комментариев. Это просто означает "снижение все в потоке входа до следующего перевода каретки". Инструкции пользователю, разъяснениям или примерам использования наиболее естественно выражены в блоке текста с каждой линией, выделенной "\".

d. Условные блоки

ANS Forth содержит IF...THEN, в виде [IF] ... [THEN].
Хотя они обычно использованы для обычной компиляции, эти слова могут быть использованы для создания блоков комментария. Таким образом, мы можем сказать

```
FALSE [IF]
[THEN]
```

и замечания, код, примеры или все, что может быть проигнорировано компилирующим механизмом.

e. Само – подтверждающий код

Устраняя неграмматические фразы как ЗАПРОС или GOSUB, Дальше пред-sents возможность - через телеграфные названия для слов - чтобы сделать кодекс почти как самодокументация и прозрачный как удобочитаемое английское или немецкое предложение. Таким образом, например, программа контроля робота могла подставить - оловянная
амальгама фраза как

```
2 TIMES   LEFT EYE   WINK
```

который является ясным (хотя это походит на ремарку для Брунхилд вампу Сигфриду). Это даже было бы возможным без большой трудности определить слова в программе так, чтобы последовательность могла быть сделана
Подобной английский языку: МИГНИТЕ ОСТАВЛЕННЫЕ ГЛАЗ 2 РАЗА. Один ключ к выполнению этого должен устранить "шумовые" слова как!,> R, и т.д., вынося их за скобки в выразительно названный - и reuseable-подпрограммы.
Другой должен организовать внесение в список подпрограммы так, чтобы это физически напомнило то, что это должно сделать. Два примера - стол скачка, определенный ниже, так же как метод для того, чтобы программировать
конечные государственные автоматы.

9. Встроенные арифметические операции

1979 или 1983 стандарты требуют, что соответствующая система Forth содержит минимум установленных определенных слов. Которые состоят арифметических операторов + - * / MOD /MOD */ для (обычно) 16-битных знаковых целых (-32767 до +32767) и эквивалентно для беззнаковых (0 to 65535), двойной длины и смешанные. Список будет обнаружен в глоссарии, сопровождающий вашу систему, также как в SF и FTR.

Попробуйте этот пример нетривиальной программы, которая использует арифметику переход для вычисления общего делителя двух целых, используя алгоритм Евклида:

```
: TUCK   ( a b -- b a b)      SWAP   OVER    ;
: GCD    ( a b -- gcd)   ?DUP   IF    TUCK   MOD       GCD   THEN   ;
```

Слово ?DUP дублирует вершину стека, если это не 0, и оставляет это только в противном случае. Если вершина стека 0, следовательно, GCD поглощает и ничего не делает. Тем не менее, если вершина стека не равно 0, затем GCD TUCKs вершину стека под NOS (чтобы сохранить); затем делит NOS, keeping the remainder (MOD). There are now two numbers left on the stack, so we again take the GCD of them. That is, GCD calls itself.

Если вы пробуете вышеуказанный код, это будет неудачно. Словарная статья не могла просмотреть и найти признак завершения строки ";", который бы завершил её. Так фактически мы должны использовать слово RECURSE, чтобы достичь успеха, как в

```
: TUCK   ( a b -- b a b)      SWAP   OVER    ;
: GCD    ( a b -- gcd)   ?DUP   IF    TUCK       MOD   RECURSE
THEN   ;
```

Теперь попытаемся

```
784 48 GCD .   <cr>   16 ok
```

Стандарты ANSI/ISO Forth (основаны в 1994) определила минимальный набор арифметических операций + - * / MOD */ /MOD */MOD и M* .

Стандартный размер текстовой памяти - ячейка, которая должна быть, по крайней мере, 16 битов, но в современных системах 32- или даже64 битов. Длина целых в Win32Forth - 32 бита. Стек в ANS Forths всегда 1 ячейка.

10. Выполнение циклов и структурное программирование

Forth имеет несколько путей образовать петли, включая неявный метод cursion, иллюстрированный выше. Рекурсия имеет дурную славу как метод перекручивания, потому что на большинстве языков, которые разрешают рекурсию, это налагает недопустимую продолжительность наверху на программу. Хуже, рекурсия может - по причинам вне возможностей этого Введения в Forth - быть чрезвычайно неэффективным методом выражения проблемы. В Forth, нет фактически никакого избытка наверху в рекурсивных обращениях, потому что Forth использует стек непосредственно.

Таким образом, нет никакой причины не повторно проклянуть, если это - лучший путь программировать алгоритм. Но в течение тех времен, когда рекурсия просто не достаточно, вот - еще некоторые стандартные методы.

a. Неопределенные циклы

Конструкция

```
BEGIN xxx ( -- flag)      UNTIL
```

Выполняет слова, представленные xxx, оставляя вершину стека в значении TRUE —в котором UNTIL завершает цикл— или FALSE —в котором UNTIL прыгает обратно к BEGIN. Попытка:

```
: COUNTDOWN        ( n --)
    BEGIN    CR    DUP   .      1 -       DUP   0    =   UNTIL
DROP    ;

5 COUNTDOWN
5
4
3
2
1   ok
```

Вариант BEGIN...UNTIL

```
BEGIN xxx ( -- flag) WHILE                yyy   REPEAT
```

Здесь xxx выполнен, WHILE тестирует значение, если FALSE окидает цикл; если TRUE, yyy выполняется; REPEAT затем возвращается к BEGIN.

Эти формы могут использоваться, чтобы настроить петли, которые повторяются, пока некоторый внешний случай (нажимающий ключ в клавиатуре, например) не заставляет значение выходить из цикла. Они могут также быть используемыми, чтобы сделать бесконечные циклы (как внешний переводчик Forth), вынуждая значение быть ЛОЖНЫМ на определении как

```
: ENDLESS           BEGIN     xxx    FALSE     UNTIL ;
```

b. Определенные циклы

Большинство Forths индексное использование циклов DO...LOOP (или +LOOP или /LOOP).
Это разрешается только в пределах определения

```
: BY-ONES          ( n --)      0 TUCK     DO    CR   DUP     .    1 +
LOOP       DROP   ;
```

Слова CR DUP . 1 + будут выполнены n раз от начала
предела, 0, увеличивается на каждом шаге до n-1.

Чтобы вести 2's, мы используем фразу 2 +LOOP перемещая LOOP, как с

```
: BY-TWOS          ( n --)      0 TUCK
      DO      CR   DUP   .    2 +       2 +LOOP         DROP     ;
```

Эти слова могут быть упрощены доступом индекса цикла словом I:

```
 : BY-TWOS        ( n --)          0   DO     CR   I   .           2
+LOOP   ;
```

Даже возможно вложение цикла и иметь доступ к внешнему индексу из внутреннего,
Используя слово J, как в

```
  : NESTED         ( n m --)     CR
           0 DO     DUP    ( n n --)
                    0 DO    CR   J .   I .
                    LOOP
           LOOP
  DROP   ;

  2 3    NESTED
  0 0
  0 1
  1 0
  1 1
  2 0
  2 1    ok
```

c. Структурное программирование

N. Wirth изобрел язык Паскаля в реакции на диаграммы процесса выполнения программы, напоминающие пластину спагетти. Такие блок-схемы часто виделись на ранних языках как ФОРТРАН и ассемблер. Вирт намеревался устранить ярлыки линии и прямые скачки (GOTOs), таким образом, вынуждая поток контроля быть ясным и прямым.

Идеал был подпрограммами или функциями, которые выполнили единственную задачу,
с уникальными записями и выходами. К сожалению, программисты настаивали на GOTOs, очень много Pascals и другие новые языки теперь имеют их. Хуже, идеал коротких

подпрограмм, которые делают одну вещь только, недостижим на таких языках, потому что метод для того, чтобы называть их и мимолетные аргументы налагает большое наверху. Таким образом, скорость выполнения заново печатные листы, минимизирующие запросы, который в свою очередь означает дольше, более сложный под - рутины, которые выполняют несколько связанных задач.

Сегодня структурированная программа - ming, кажется, означает немного больше чем письмо кода с вложенной МЕЖДУНАРОДНОЙ ФЕДЕРАЦИЕЙ, заказанной симпатичным принтером.

Как это ни парадоксально, Forth является единственным действительно структурированным языком, вместе используют, хотя это не было разработано с этим как его цель. В Forth слове определения – списки подпрограмм. Язык не содержит никакого GOTO'S, таким образом, невозможно написать код "спагетти". Forth также поощряет структуру через короткие определения. Дополнительная продолжительность, понесенная в ломке длинной процедуры на многие маленькие, (это называют, "выступая в качестве фактора") является типично довольно маленькой в Forth. Каждый Forth подпрограмма (слово) имеет один вход и один пункт выхода, и может быть написана, чтобы выполнить единственную работу.

d. «Нисходящее» программирование

"Нисходящее" программирование - доктрина, которую нужно проектировать всю программу детально:

> Делают схему, блок-схему или что бы то ни было, беря широкий краткий обзор целой проблемы.

> Ломают проблема на маленькие части (анализируйте это).

> Тогда кодируют индивидуальные компоненты.

Естественный программный способ в Forth - "восходящая" а не "вершина - вниз" – самое общее слово кажется последним, тогда как определения должны прогрессировать от примитива до комплекса. Это приводит к некоторым, - какой различный подход с более знакомых языков:

> В Forth компоненты определены примерно, и затем, поскольку они закодированы, они немедленно проверены, отлажены, перепроектированы и улучшены.

> Развитие компонентов ведет развитие внешних уровней программы.

11. CREATE ... DOES> (the pearl of FORTH)

Michael Ham has called the word pair CREATE...DOES>, the “pearl of Forth”. CREATE is a component of the compiler, whose function is to make a new dictionary entry with a given name (the next name in the input stream) and nothing else. DOES> assigns a specific run-time ac-tion to a newly CREATEd word.

a. Определение “определяющих” слов

CREATE находит свое наиболее важное использование в мощный класс слов Forth, называемых “определяющими” словами. Компилятор двоеточия ":" такое слово, как VARIABLE и CONSTANT.

Определение VARIABLE в высокоуровневом Forth - простое

```
: VARIABLE   CREATE    1 CELLS     ALLOT ;
```

Мы уже видели, как VARIABLE был использован в программу. (Вот альтернативное определение в Forths

```
: VARIABLE   CREATE    0   ,   ;
```

—эти переменные инициализированы в 0.)

Forth позволяет нам определять инициализированные слова для содержания специфических величин:

например, мы захотели определить число 17 словом. CREATE и "," ("запятая") можно сделать так:

```
17 CREATE SEVENTEEN    ,   <cr>    ok
```

Теперь протестируйте это

```
SEVENTEEN @ .   <cr>   17 ok .
```

Замечания:

> Слово, ("запятая") устанавливает вершину стека в следующую ячейку словаря и увеличивает указатель словаря этим количеством байтов.

> Слово "C," ("смотри запятую") существует также — он устанавливает символ на символьную длину следующей щели словаря и увеличивает указатель на 1.

b. Время прогона против времени компиляции действий

В предыдущем примере, мы инициализировали переменную SEVENTEEN в 17, когда мы создавали, но мы все ещё должны выбирать из стека переменную SEVENTEEN @ всякий раз, когда мы хотели бы. Это не совсем то, что мы имели в виду: мы хотели бы найти 17 в вершине стека когда SEVENTEEN названа. Слово DOES> дает нам средство для этого.

Функция DOES> должна определить время прогона действия для определяющих слов “child”. Рассматривайте определяющее слово CONSTANT , определенного на высоком уровне (конечно, CONSTANT обычно определено в машинном коде для скорости) Forth

```
: CONSTANT    CREATE    ,   DOES>   @   ;
```

и использовано как

```
53 CONSTANT PRIME      <cr> ok
```

Теперь тестируем:

```
PRIME . <cr>    53   ok .
```

Что здесь происходит?

> CREATE (спрятан в CONSTANT) создает входную названную PRIME (первое слово во входном потоке следующим за CONSTANT). Затем "," места в вершине стека (число 53) в следующей ячейке словаря.

> Затем DOES> (внутри CONSTANT) добавляет действия всех слов между ним и ";" (конец определения) —в этом случае, "@"— в слове, определенным CONSTANT.

c. Размерные данные (существенные устройства)

Вот - пример власти определения слов, и различия между временем компиляции и временем выполнения.
Физические проблемы вообще вовлекают количества, которые имеют измерения, обычно выражаемые как масса (M), длина (L) и время (T) или другие продукты. Иногда есть больше чем одна система единиц, которых используют вместе, чтобы описать те же самые явления.

Например, американская или английская полиция, сообщающая о несчастных случаях
могла бы использовать дюймы, фунты и ярды; в то время как Континентальная полиция использовала бы сантиметры и метры. Вместо того чтобы писать различные версии программы анализа несчастного случая, более просто написать одну программу и сделать
единицу преобразования части грамматики. Это легко в Forth.

Самый простой метод должен держать все внутренние длины в миллиметрах,

скажем, и конвертируемом следующим образом:

```
: INCHES   254        10   */ ;
: FEET     [ 254 12 * ] LITERAL         10   */ ;
: YARDS    [ 254 36 * ] LITERAL         10   */ ;
: CENTIMETERS         10   * ;
: METERS       1000    * ;
```

Замечание: Этот пример основан на арифметике целых чисел.
Слово */ означает “умножать третье число в стеке NOS, сохраняя двойную точность, и делятся TOS”. То есть, the комментарии стека для */ - (a b c -- a*b/c).

Использование должно быть

```
10 FEET    .   <cr>    3048 ok
```

Слово "[" включает из режима compile в режим interpret при компиляции. (Если система интерпретирует эти изменения.) Слово "]" включает из режима interpret в режим compile mode.

Исключая некоторую ошибку-проверки, “определение” компилятора двоеточия ":" - просто

```
:   :     CREATE   ]     DOES>     doLIST   ;
```

то же самое с ";" - просто

```
:   ;     next       [    ;   IMMEDIATE
```

Другое использование для этих выключателей должно выполнить арифметику во время компиляции, а не во время выполнения, и для ясности программы и для простой модификации, поскольку мы сделали в первой попытке в проставленных размерах данных (то есть, фразы, типа

```
[ 254 12 * ] LITERAL
```

и

```
[ 254 36 * ] LITERAL
```

который позволил нам включать ясным способом число десятая части в футе или ярде.

Предыдущий метод контакта с модулями требовал излишне многих определений и генерировал ненужный код. Более компактный подход использует слово определения, МОДУЛИ:

```
: D,    ( hi lo --)             SWAP   , ,     ;
: D@    ( adr -- hi lo)             DUP   @        SWAP   CELL+
@   ;
: UNITS      CREATE       D,      DOES> D@      */ ;
```

Затем мы могли бы сделать таблицу

```
254 10                UNITS INCHES
254 12 *         10   UNITS FEET
254 36 *         10   UNITS YARDS
10   1                UNITS CENTIMETERS
1000     1            UNITS METERS

\ Usage:

10 FEET       . <cr>      3048    ok
3 METERS . <cr>           3000    ok
\ .....................
\ etc.
```

Это улучшение, но Forth разрешает простое расширение, которое допускает преобразование на входные устройства, для использования в выводе:

```
VARIABLE       <AS>           0 <AS> !
: AS       TRUE   <AS> ! ;
: ~AS      FALSE <AS> ! ;
: UNITS    CREATE     D,   DOES>       D@   <AS> @
   IF   SWAP    THEN
   */     ~AS      ;

\ UNIT DEFINITIONS REMAIN THE SAME.
\ Usage:
10 FEET    .    <cr>    3048     ok
3048 AS FEET      .   <cr>      10   ok
```

d. Передовое использование компилятора

Предполагайте, мы имеем серии кнопок, пронумерованных от 0-3, и слово WHAT для их чтения. То есть, WHAT введения с клавиатуры: когда кнопка #3 нажата, например, WHAT оставляет 3 в стеке.

Мы хотели бы определить слово BUTTON для выполнения действия проталкивания n'th кнопки, так, что мы могли бы просто сказать:

```
WHAT BUTTON
```

На стандартном языке BUTTON приблизительно выглядит так

```
: BUTTON    DUP   0 =      IF    RING     DROP    EXIT   THEN
   DUP   1 =      IF    OPEN     DROP    EXIT   THEN
    DUP   2 =      IF    LAUGH DROP       EXIT   THEN
    DUP   3 =      IF    CRY      DROP    EXIT   THEN
    ABORT" WRONG BUTTON!"            ;
```

Таким образом, мы должны были бы пройти два решения в среднем.

Forth делает возможным намного более опрятный алгоритм, вовлекая "таблицу переходов". Механизм, которым Forth выполняет подпрограмму, должен подать ее "лексему выполнения" (часто адрес, но не обязательно) к слову EXECUTE. Если мы имеем таблицу лексем выполнения, мы должны только искать тот, соответствующий индексу (смещение в таблицу) выбирают это к стеку и говорят, EXECUTE.

Один путь, чтобы закодировать это

```
CREATE   BUTTONS   ' RING ,    ' OPEN ,     ' LAUGH ,   ' CRY ,
: BUTTON    ( nth --)      0 MAX    3 MIN
        CELLS   BUTTONS   +   @   EXECUTE   ;
```

Отметьте, как фраза 0 MAX 3 MIN защищает против диапазона индекса. Хотя Forth философия не должна замедлить код с ненужной проверкой ошибок (потому что слова проверены, поскольку они определены), программируя пользовательский интерфейс, некоторая форма обработки ошибок жизненно важна. Обычно проще предотвратить ошибки, поскольку мы только сделали, чем предусмотреть восстановление после того, как они сделаны.

Как метод таблицы действия работает?

> CREATE BUTTONS делает BUTTONS входа словаря.

> слово '("импульс сигнала времени") находит лексему выполнения (xt) следующего слова, и слово, ("запятая") хранит это в области данных нового слова BUTTONS. Это повторено до всех подпрограмм, среди которых мы хотим выбрать, хранили их xt's в таблице.

> BUTTONS таблица теперь содержит xt's различных действий BUTTON.

> CELLS тогда умножает индекс соответствующим числом байтов в ячейку, получить смещение в таблице BUTTONS желательно

xt.

> BUTTONS + тогда добавляют, что базовый адрес BUTTONS, чтобы получить абсолютный адрес, где xt сохранен.

> @ выбирает xt для EXECUTE,чтобы выполниться.

> EXECUTE тогда выполняет слово, соответствующее нажатой кнопке. Просто!

Если программа нуждается только одна таблица действия, предыдущий метод достаточен.

Однако более сложные программы могут требовать многих таких. В том случае это может заплатить, чтобы установить систему для того, чтобы определить таблицы действия, включая и предотвращающий ошибку код и код, который выполняет надлежащий выбор. Один способ закодировать это

```
: ;CASE      ;                                    \ do-nothing word
: CASE:
    CREATE        HERE      -1   >R        0   ,      \ place for length
    BEGIN         BL    WORD     FIND                 \ get next subroutine
        0=     IF       CR    COUNT        TYPE     ."   not found"   ABORT
THEN
        R>     1+      >R
        DUP       ,      [']  ;CASE        =
    UNTIL         R>     1-       SWAP      !         \ store length
    DOES>         DUP    @        ROT                 ( -- base_adr len n)
                  MIN    0    MAX                     \ truncate index
                  CELLS      +   CELL+         @   EXECUTE   ;
```

Отметьте две формы проверки ошибки. Во время компиляции, CASE: Компиляция отказов нового слова, если мы спрашиваем для указания на неопределенную подпрограмму:

```
case: test1           DUP    SWAP      X   ;case
     X not found
```

и мы считаем, сколько подпрограмм в таблице (включая пустую) и ищем индекс в диапазоне [0,n].

```
CASE:   TEST       *    /    +   -     ;CASE       ok
15 3 0 TEST . 45           ok
15 3 1 TEST . 5          ok
15 3 2 TEST . 18           ok
15 3 3 TEST . 12           ok
15 3 4 TEST . . 3 15            ok
```

Просто для разнообразия, здесь иной путь того, что мы сделали:

```
: jtab:     ( Nmax --)              \ starts compilation
   CREATE                          \ make a new dictionary entry
   1-    ,                         \ store Nmax-1 in its body
;                                  \ for bounds clipping

: get_xt          ( n base_adr -- xt_addr)
     DUP     @            ( -- n base_adr Nmax-1)
     ROT                  ( -- base_adr Nmax-1 n)
     MIN     0    MAX          \ bounds-clip for safety
     1+    CELLS+         ( -- xt_addr = base + 1_cell + offset)
;

: |     '   ,   ;       \ get an xt and store it in next cell

: ;jtab     DOES>     ( n base_adr --)     \ ends compilation
                get_xt     @   EXECUTE         \ get token and execute it
      ;     \ appends table lookup & execute code

\ Example:
```

```
 : Snickers      ." It's a Snickers Bar!"       ;   \ stub for test

      \ more stubs

      5 jtab:     CandyMachine
                  | Snickers
                  | Payday
                  | M&Ms
                  | Hershey
                  | AlmondJoy
      ;jtab

      3 CandyMachine       It's a Hershey Bar!     ok
      1 CandyMachine       It's a Payday!   ok
      7 CandyMachine       It's an Almond Joy!     ok
      0 CandyMachine       It's a Snickers Bar!    ok
            -1 CandyMachine          It's a Snickers Bar!    ok
```

12. Плавающая точка арифметики

Хотя Forth однажды сторонится с плавающей запятой арифметика (потому что в прежнюю эру, чем математическая арифметика целого числа сопроцессоров была 3x быстрее), в последние годы стандартный набор названий слова был согласован. Это разрешает обмен программами, которые будут работать правильно на любом компьютере, так же как развитии Научной Библиотеки Подпрограммы в Forth (FSL).

Хотя Стандарт ANS не требует отдельного стека для с плавающей запятой чисел, большинство программистов, которые используют Forth для числового анализа, использует отдельное, с плавающей запятой располагают в стеке; и большинство подпрограмм в FSL принимает такой. Мы сделаем, таким образом, здесь также.

С плавающей запятой операторы имеют следующие названия и выполняют действия, обозначенные в сопроводительных комментариях стека:

```
F@      ( adr --)     ( f: -- x)
F!      ( adr --)     ( f: x --)
F+                    ( f: x y -- x+y)
F-                    ( f: x y -- x-y)
F*                    ( f: x y -- x*y)
F/                    ( f: x y -- x/y)
FEXP                  ( f: x -- e^x)
FLN                   ( f: x -- ln[x])
FSQRT                 ( f: x -- x^0.5)
```

Дополнительные операторы, функции, тригонометрические функции, и т.д. могут быть найдены в FLOATING и FLOATING EXT текстовых комплектах. (См. dpans6- доступный в HTML, Postscript и форматах MS Word. К версии HTML можно обратиться от этой домашней страницы.)

Чтобы помогать в использовании с плавающей запятой арифметике, я создал простой
ПОДОБНЫЙ FORTRAN интерфейс для того, чтобы включить формулы в Forth слова.

Файл ftest.f (включенный ниже) иллюстрирует, как ftran201.f должен использоваться.

```
\ Тест для транслятора формул ANS

marker -test
fvariable a
fvariable b
fvariable c
fvariable d
fvariable x
fvariable w

: test0   f" b+c"     cr   fe.
      f" b-c"   cr    fe.
      f" (b-c)/(b+c)"       cr fe.   ;
```

```
3.e0 b f!
4.e0 c f!
see test0
test0

: test1     f" a=b*c-3.17e-5/tanh(w)+abs(x)"      a f@   cr fe.   ;
      1.e-3 w f!
      -2.5e0 x f!
cr cr
see test1
test1

cr cr

: test2     f" c^3.75"     cr fe.
      f" b^4"        cr fe.   ;
see test2
test2

\ Baden's test case

: quadroot c f! b f! a f!
  f" d = sqrt(b^2-4*a*c) "
  f" (-b+d)/(2*a) "     f" (-b-d)/(2*a) "
;

cr cr
see quadroot

: goldenratio    f" max(quad root(1,-1,-1)) "      ;

cr cr
see goldenratio
cr cr

goldenratio f.

0 [IF]
```

Вывод выглядит так:

```
: test0
      c f@ b f@ f+ cr fe. c f@ fnegate b f@ f+ cr fe. c f@ fnegate b f@
      f+ c f@ b f@ f+ f/ cr fe. ;

7.00000000000000E0
-1.00000000000000E0
-142.857142857143E-3

: test1
      x f@ fabs 3.17000000000000E-5 w f@ ftanh f/ fnegate b f@ c f@ f* f+
      f+ a f! a f@ cr fe. ;

14.4682999894333E0    ok

: test2
      c f@ noop 3.75000000000000E0 f** cr fe. b f@ f^4 cr fe. ;

181.019335983756E0
81.0000000000000E0    ok

: QUADROOT        C F! B F! A F! B F@ F^2 flit 4.00000 A F@
            C F@ F* F* F- FSQRT D F! B F@ FNEGATE D
            F@ F+ flit 2.00000 A F@ F* F/ B F@ FNEGATE
            D F@ F- flit 2.00000 A F@ F* F/ ;

: GOLDENRATIO              flit 1.00000 flit -1.00000 flit -1.00000
            QUADROOT FMAX ;
```

```
      1.61803    ok
```

with more or fewer places.

[THEN]

13. Нетривиальный пример программирования

Чтобы иллюстрировать, как создать нетривиальную программу, позвольте нам разрабатывать корневое средство двоичного поиска. Мы будем использовать ТРАНСЛЯТОР ФОРМУЛЫ ftran201.f, чтобы упростить появление кода (то есть, это скрывает выборки данных, которые иначе требовались бы).

Сначала мы должны понять алгоритм полностью:

Если мы знаем, что корни заключены в скобки между xa и xb, и что f (xa) *f (xb) <0 (по крайней мере 1 корень находится в интервале) мы берем следующее предположение, что xp = (xa+xb)/2.

Мы тогда оцениваем функцию в xp: fp = f (xp).
Если fa*fp> 0 мы устанавливаем xa = xp, еще мы устанавливаем xb = xp.
Мы повторяемся, пока концы интервала, содержащего
корень не достаточно близко друг к другу.

Чтобы начинать программировать, мы отмечаем, что мы должны будем следить за тремя пунктами: xa, xb и xp. Мы также должны следить за тремя функциональными значениями, оцененными в тех пунктах, Pa, Rb и Rp. Мы также должны определить точность, эпсилон, в пределах которого мы ожидаем определять корень.

Затем мы должны определить пользовательский интерфейс. Таким образом, как только мы имеем подпрограмму, которая находит корни, как мы вызовем это?
Так как мы хотели бы в состоянии определить название функции, чтобы найти корень, в то же самое время мы определяем интервал, мы думаем, что корень находится внутри, мы нуждаемся в некотором способе передать название к корневому средству поиска как параметр.

Я предварительно разработал интерфейс, который удовлетворяет этому:
я говорю

```
      use( fn.name xa xb precision )bin_root
```

как в

```
use( f1   0e0 2e0 1e-5 )bin_root
```

и корень будет оставлен в стеке.

Код для переходящих имен функций, как аргументы включены, когда вы загружаете ftran201.f — слова использованы в этой программе – использование bin_root (, v: и определяет . v: создает ложную словарную статью, которая может быть сделана для выполнения фактической функции).

Вот структура данных и их идентификаций:

```
MARKER -binroots     \ say -binroots to unload

\ Data structures

FVARIABLE Ra                        \ f(xa)
FVARIABLE Rb                        \ f(xb)
FVARIABLE Rp                        \ f(xp)
FVARIABLE xa                        \ lower end
FVARIABLE xb                        \ upper end
FVARIABLE xp                        \ new guess
FVARIABLE epsilon                   \ precision

v: dummy                           \ create dummy dictionary entry
```

Корневое обнаружение фактической подпрограммы,)bin_root , будут совсем просты и легко для следования:

```
: )bin_root   ( xt --)   ( f: Low High Precision -- root)
      initialize
      BEGIN   NotConverged?   WHILE   NewPoint    REPEAT
      f" (xa+xb)/2"           ( f: -- root)
;
```

Отметьте, что подпрограммы, включающие это по телеграфу называют так, они не нуждаются ни в каком объяснении; тогда как)bin_root объясняется в соответствии с его комментариями стека.

Комментарии к первой линии указывают, что)bin_root ожидает "лексему выполнения" на стеке данных, и три с плавающей запятой числа на с плавающей запятой стек. Они - его параметры. (См. 11d для обсуждения), Лексема выполнения - то, что используется, чтобы изменить поведение фиктивной dummy входа словаря: мы говорим,

```
defines dummy
```

в слове, инициализируют, чтобы заставить dummy вести себя как функция, корень которой мы ищем.

Конечный комментарий (f: - корень), указывает, что))bin_root оставляет ответ на с плавающей запятой стек.

В некотором смысле мы программируем от вершины вниз, так как мы начали с последнего определения программы и работаем дальше.

В Forth мы часто идем по обоим путям - вершина вниз и основание наверх – в то же самое время.

Ключевые слова, которые мы должны теперь определить, - initialize, NotConverged? и NewPoint. Мы могли бы также начать, инициализируют, так как это концептуально просто:

```
: initialize    ( xt --) ( f: lower upper precision --)
    defines dummy                          \ xt -> DUMMY
    f" epsilon="    f" xb="     f" xa="    \ store parameters
    f" Ra=dummy(xa)"
    f" Rb=dummy(xb)"
    f" MoreThan( Ra*Rb, 0)"                \ same sign?
    ABORT" Even # of roots in interval!"
;
```

Слово ABORT"печатает сообщение, которое следует за этим и прерывает выполнение, если это сталкивается с флажком TRUE на стеке данных. Это широко используется как простой ошибочный обработчик.

ABORT (без") просто прерывает выполнение, когда с этим сталкиваются. Таким образом, это обычно находится в некоторой структуре решения как IF...THEN пункт. (См. 11d для двух примеров использования.)

ABORT" предшествовал тест. Чтобы использовать тест как функцию в выражении Fortran-like (этот тест потребляет два параметра от с плавающей запятой стек и оставляет флажок на стеке данных), мы должны определить синоним для этого. Причина - то, что ftran201.f не распознает реляционные операторы подобно > или < . Определение*

```
: MoreThan     ( f: a b)    ( -- true if a>b)
    POSTPONE   F>   ;   IMMEDIATE
```

Код произведенный f" MoreThan(Ra*Rb, 0)" затем только

```
RA F@ RB F@ F* flit 0.00000E-1 F>
```

который является тем, что мы хотим. Мы уже объяснили, что фраза defines dummy. Фразы f" xa =" и так далее являются стенографией для того, чтобы хранить кое-что от плавающей запятой стек с плавающей запятой переменная. Таким образом, f" xa ="

генерирует код XA F!. Остальная часть initialize должна вычислить функцию в оконечных точках воображаемого интервала ограничения (a, b).

NotConverged? является тестом на (не) конвергенция. WHILE ожидает флажок на стеке данных, как описано в 10a. Таким образом, мы определяем

```
: NotConverged?       ( -- f)
    f" MoreThan( ABS( xa - xb ), epsilon )"      ;
```

которая генерирует код

```
XB F@ XA F@ F- FABS EPSILON F@ F>
```

Что насчет NewPoint ? Ясно,

```
: NewPoint
    f" xp = (xa+xb)/2"          \ new point
    f" Rp = dummy(xp)"
    f" MoreThan( Ra*Rp, 0)" \ xp on same side of root as xa?
    IF        f" xa=xp"   f" Ra=Rp"
    ELSE      f" xb=xp"   f" Rb=Rp"   THEN
;
```

Таким образом, мы генерируем новое предположение делением пополам, оцениваем функцию там и решаем, как выбрать новый интервал ограничения.

Все, что остается, должно поместить определения надлежащий заказ и проверить результат, загружая программу bin_rts.f и испытывая случай.

```
FALSE [IF]
```

Пример:

```
: f1     fdup   fexp   f*   1e0   f-   ;   ok
use( f1    0e0 2e0 1e-5 )bin_root f. .567142    ok
```

```
[THEN]
```

Наконец, если мы хотим быть очень осторожным действительно, и/или планируем многократно использовать программу, мы добавляем соответствующий шаблонный заголовок, типа включенного в файл

bin_rts.f .

*Отметьте: слово POSTPONE в этих средствах контекста, что слово после этого –
в этом случае F> - будет собрано в слово, которое использует MoreThan, а не в MoreThan.
(Отметьте, что MoreThan НЕПОСРЕДСТВЕНЕН.) Этот способ сделать вещи сохраняет некоторых наверху в течение выполнения. Некоторые Forths (особенно Win32Forth) определяют слово SYNONYM, чтобы достигнуть той же самой вещи.

14. Некоторые замечания о строках

Как в других языках, алфавитно-цифровые строки в Forth представлены как непрерывные массивы в памяти, каждый модуль памяти, являющийся "символом".
Традиционно символ, закодированный ASCII or EBCDIC расширенного двоично-десятичного кода обмена информацией занял один (1) байт памяти, учитывая 256 символов. С потребностью закодировать алфавиты кроме латинского (например, Китайский, арабский, иврит, Кириллица) двух-байтовое кодирование по имени Unicode было принято, который учитывает 65535 отличных символов.

Традиционная Forth строка состояла из байта счета и до 255 байтов, содержащих
алфавитно-цифровые символы (обычно в ASCII). В ANS Forth была оставлена эта схема:
то, как строки сохранены, будет зависеть от выполнения. Однако ANS Forth содержит слова, которые дают возможность нам управлять строками независимо от того, как они осуществлены.

Большинство ANS Forths (и Win32Forth - один из них) определяет S", чтобы определить интерпретирующее так же как собирающее поведение. Это означает это, если мы говорим

```
S" This is a string!" CR TYPE
```

мы получаем

```
This is a string! ok
```

Что случилось? S" This is a string!" создана строка, начиная текст с “c-address” и с “count”, которое сообщает, как много символов (включая пробелы) строка содержит. Адрес и счет оставлены в стеке. То есть, соответствующее изображение стека должно быть

```
S" This is a string!"    ( -- c-addr u)
```

(счет является беззнаковым целым u, поскольку строки отрицательной длины бессмысленны).

Слово CR означает “insert a carriage return”, и TYPE означает “from the c-addr output u characters to the screen”.

Задание:
Используйте, что вы уже выучили для написания программы “Hello world!”.

Совершенно выполнимо определить собственный набор слов для того, чтобы работать со строками в зависимости от того, какое приложение каждый пишет. Например, я написал программу, чтобы транслировать математические формулы подобно форме Fortran в коде Forth, выводя любой результат на экран (в испытательных целях) или внедряя это в определение Forth. Есть даже вариант, который оценивает формулу, если все переменные были предварительно определены и даны числовые значения.
Чтобы достичь это, требовалось строк длиннее чем 255 символов, таким образом, я определил мое собственное.

Я теперь хочу обратиться "к изображенному числовому выводу". Много компьютерных программ должны вывести числа в некотором специфическом формате, независимо от того как они сохранены интерн – союзник. Например, программа учета могла бы вывести денежно-кредитные количества в обычном формате долларов-центов. Вот слова, которые достигают этого,

```
# ,   <# ,    #S ,    #> , SIGN и HOLD
```

Они не имеют никакого определенного интерпретирующего поведения (хотя нет никакого сообщения, что любая деталь Forth может сделать), и предназначены, чтобы использоваться в пределах определений слова.
Вот - пример: предположите, что мы пишем программу учета. Так как большинство пользователей не будет иметь дело с количествами, которые превышают 100 000 000 $, мы можем использовать подписанные 32- разрядные целых числа, чтобы представить доллары и центы. (Такие числа могут представить количества до ±(231—1) = ±2147483647 cents центов.)
Подписанные целые числа двойной длины, по крайней мере, 32 бита длиной на всех СОВМЕСТИМЫХ С ANS системах (хотя они будут 64 бита на 32-разрядных компьютерах). Следовательно, мы будем использовать удвоение так, программа выполнится на любом СОВМЕСТИМЫЙ С ANS Forth.

Целое число двойной длины введено от клавиатуры включением десятичной точки в этом, как

```
-4756.325     ok
```

Давайте определим слово двойной длины на выход. Первая часть должна переводиться в текстовую строку, отосланную c-addr u.

```
: (d.$) ( d -- c-adr u) TUCK DABS  <#  # # [CHAR] . HOLD #S ROT SIGN  #> ;
```

Так как комментарий стека (d -- c-adr u) показывает, (d.$) поглощает целое двойной длины из стека и оставляет данные строки в форме, которая может быть

напечатана на экране словом TYPE. Давайте протестируем:

```
4376.58    (d.$)     CR    TYPE
4376.58 ok

-4376.99     (d.$)    CR    TYPE
-4376.99 ok
```

Стоит исследовать то, что делает каждая часть. Двойное целое число длины сохранено как две ячейки на стеке, с наиболее значительной частью на ВЕРШИНЕ СТЕКА. Таким образом, слово TUCK места наиболее значительная часть (содержащий алгебраический признак) выше d, и затем DABS новообращённых d к |d|. Затем, <# начинает процесс построения алфавитно-цифровой строки.

Два случая # снимают с этих двух младших разрядов и помещают их в строку. Фраза [CHAR].

HOLD добавляет десятичную точку к строке.

[CHAR] встраивает в представление символа . как числовая опечатка (в ASCII это - 46).

HOLD тогда добавляет это к строке в работе. (HOLD, не имеет никакого значения кроме между <# и #>.)

Тогда слово #S берет остальную часть цифр и добавляет их к возникающей строке.

(Полу) наконец, ROT помещает наиболее значительную часть d (с его признаком) на ВЕРШИНЕ СТЕКА, и SIGN добавляет его алгебраический признак к началу строки. (Снова, SIGN только значим между <# и #>.)

И наконец, слово #> очищает все и оставляет c-addr u на стеке, готовом к дисплею
или что бы то ни было.

Задания:

a) Как вы добавите знак доллара ($) в выходной номер?
b) Как вы приложите отрицательную сумму в круглых скобках без отображения знака —? [То есть (4376.99), а не -4376.99.]
c) Определите слово для отображения числа в формате доллара и цента.

15. Некоторые полезные ссылки

> M. Kelly and N. Spies, Forth: A text and Reference (Prentice-Hall, NJ, 1986)
> L. Brodie, Starting Forth, 2nd ed. (Prentice-Hall, NJ, 1986)
> L. Brodie, Thinking FORTH (Prentice-Hall, NJ, 1984 (online edition)